Streamed Watershed Transform on GPU for Processing of Large Volume Data

Michal Hučko
Comenius University, Bratislava, Slovakia

Miloš Šrámek
Austrian Academy of Sciences, Vienna, Austria

Abstract

Since its introduction the watershed transform became a popular method for volume data segmentation. A range of various algorithms for its computation were developed, including parallel algorithms for computation on different architectures. Recently also algorithms for consumer graphical accelerators were developed. Neither of these, however, are able to process data larger than the available memory as the whole data has to be present in the memory of the device. In this paper we present two versions of a streamed multi-pass algorithm for watershed computation on a GPU. As the slice-based streaming approach is used both variants are capable of processing data exceeding the size of the available graphics accelerator memory.

CR Categories: I.4.0 [Computing Methodologies]: Image Processing and Computer Vision—General

Keywords: watershed transform, slice-based processing, volume data, parallel processing, OpenCL

1 Introduction

Segmentation is one of the often performed volume data processing tasks. There exists a range of methods for volume data segmentation with watershed transform being one of the commonly used one. The method was originally proposed by Digabel and Lantuejoul [Digabel and Lantuejoul 1978] and has its origins in mathematical morphology. Since then various algorithms were developed – a thorough overview, analysis and comparison can be found in [Roerdink and Meijster 2000]. A reader interested in detailed description and definition of watershed transform should inspect the mentioned paper.

In this paper we present algorithms specially developed for processing of large volume data. The watershed transform, being a global operation, in general requires access to the whole volume in order to process a single voxel. Easily satisfied when the volume data fits into memory of device where the processing is performed, this requirement poses a problem when the data is larger. Common algorithms for the watershed transform and their available implementations usually do not consider data sizes exceeding limits of the available memory and therefore cannot often be used. In this paper we present a stream-based algorithm which, owing to its small memory footprint, is able to process arbitrary large data. Both CPU and GPU variants are described and compared.

2 The watershed transform

The watershed transform is analogous to division of a geographical landscape to catchment areas of different lakes/seas/oceans. For 2D images this could be visualised by extruding pixels into the third dimension by their pixel value. Pixels with higher value would form ridges, whereas pixels with smaller values will form valleys. Local minima in the landscape represent catchment basins and the pixels are classified by tracing in which catchment basin rain-water falling onto pixels flows.
Another common intuitive definition is based on flooding of the landscape by underground water which starts to form lakes in local minima. As the water level rises and some lakes are about to merge, this is prohibited by building dams – watershed lines. A result of a watershed transform can be seen on image 2.

Figure 2: Watershed transform of a CT data set. Borders between regions are in white.

There exist various formal definitions whose overview can be found in [Roerdink and Meijster 2000]. We will use the definition with local condition [Bieniek et al. 1997] which does not produce watersheds – lines separating neighbouring regions – directly, but instead classifies all voxels as belonging to some basin. This, without introducing any limitations, simplifies the computation and subsequent parallelisation.

Consumer graphical hardware, while providing unparalleled computational power when compared to CPUs, lacks capabilities of performing more complex code and has limits in available registers and local memory. This makes direct parallelisation of common algorithms problematic. Recently, however, algorithms were published designed specially for current graphical accelerators. Kauffmann and Piché [Kauffmann and Piché 2008; Kauffmann and Piché 2010] presented a cellular automaton executed on a GPU, iteratively processing the data until the required result is obtained. Another two algorithms were proposed by Körbes et al. [Körbes et al. 2009] which perform also steps which have to be otherwise done in preprocessing – local minima localisation and labelling and processing of plateaus. Also other authors presented their version of the watershed transform for a GPU [Pan et al. 2008; Wagner et al. 2009]. Recently Körbes et al. [Körbes et al. 2011] compared their method with the previously mentioned cellular automaton.

All aforementioned methods expect data to fit into the graphical memory not providing means to process larger data.

3 Slice-based watershed algorithms

When data exceeds memory capacity of a device where the computation should be performed it has to be divided into smaller parts [Law et al. 1999]. For local operations – ones which in order to process a voxel access only a limited neighbourhood of the voxel – division of the volume into bricks and their processing one by one is the common praxis. The watershed transform being a global operation may generally require to visit any voxel in the volume in order to process a voxel. This would result in repetitive loading and swapping of bricks into the memory resulting in poor execution times.

Another option available for local operations is slice-based streaming where data is processed slice-by-slice. Varchola et al. [Varchola et al. 2007] presented a set of command-line tools which implement various point and local operations in such slice-based fashion. To process a slice a certain number of neighbouring slices has to be present in the memory providing the necessary neighbourhood for the voxel in the processed slice. The number of neighbouring slices depends on the performed operation and its parameters. After a slice is processed the result can be output, the oldest slice in the neighbourhood released as it won’t be needed any more and one new slice loaded into the memory to complete the neighbourhood of the next unprocessed slice. This way the whole volume can be processed without the need to actually load the whole volume into memory at once.

We developed our watershed algorithms for the f3d suite [Šrámek et al. 2004] which consists of a set of command line applications performing various simple operations on data in a filter fashion – data is read from the standard input, processed and output to the standard output. Within this framework a tool for watershed transform would merely read on its input preprocessed data produced by other tools and output the watered data via its standard output for storage or further processing. In this sense the required tools include:

- data smoothing
- gradient computation
- local minima detection
- local minima labelling (including plateau detection)
- watershed transform using the gradient and the minima label volumes

From all these tasks we inspect here only the last step since all other operations and their streamed implementations are available in the f3d package. The data is smoothed with the Gaussian filter for which also a multi-pass version with constant memory demands ([Young and van Vliet 1995]) exists if width of the filter would be too high to cause problems with memory storage.

3.1 A load-on-demand algorithm

The f3d suite originally contained a version of the watershed transform which tried to minimize the required memory by loading slices ‘on demand’. Here, a slice is loaded and processed voxel by voxel. For each voxel a path of the steepest descent is followed until a local minimum is found. When the path would lead into a slice not yet present in the memory this is at that moment loaded. After a local minimum is encountered, all voxels on the path are labelled with the local minimum label. After the whole slice is processed it is output and its representation in memory is freed, as it will not be needed anymore.

The watershed transform generally produces highly over-segmented results as each local minimum in the data forms a basin. To cope with this the data might be smoothed in preprocessing. The more smoothed the data the larger the resulting regions are. Memory demands of the aforementioned algorithm are therefore strongly data-dependent as the maximum number of slices which need be present in the memory at some time is directly related to dimensions of the largest region.

3.2 A multi-pass CPU algorithm

To cope with the uncertainty of the amount of required memory for the load-on-demand algorithm we already developed a multi-pass algorithm with stable memory requirements (data independent, only slice size matters) [Hučko and Šrámek 2011]. This is
Three neighbouring slices are present in the memory (with the exception of the start and end slices of the volume). The middle one is processed voxel by voxel. For each voxel a path of the steepest descent is followed. This path may lead to a previous (already processed) slice. In that case either an already labelled voxel is encountered or a pointer pointing to the current slice is found (the previous slice satisfies Condition 1). Otherwise, after a certain number of steps we either find a labelled voxel or the path leads to the next (not yet processed) slice. We store either value into all voxels of the path. After processing of all voxels in the middle slice this slice fulfils Condition 1.

Pass 2 We process the volume in the backward direction, starting with the last slice. As it fulfils Condition 1 all voxels are labelled with some local minimum label (there is no next slice). Now the processing loads the next slice (previous in the sense of the direction of Pass 1) of the volume and labels all voxels which contain pointers to the previous fully labelled slice making also this slice fully-labelled. After that this slice can be stored out-of-core and freed. The processing continues in this way until the whole volume is processed.

Pass 3 As the volume was read and stored in Pass 2 backwards it is necessary to flip it once more. Output from Pass 2 is therefore read backwards (i.e., in the original forward direction) and finally written to output.

A detailed comparison of the load-on-demand and multi-pass algorithm was presented in [Hučko and Šrámek 2011].

4 Slice-based GPU watershed algorithms

The implementation of a load-on-demand slice-based watershed algorithm on GPU is problematic as it requires adaptive growth of the allocated memory. This cannot be achieved without copying data between buffers at each reallocation which would slow the computation down. The load-on-demand method also has higher memory requirements which in the case of GPU implementation may represent a more serious problem than for a CPU-based one.

Therefore, we will consider only the multi-pass algorithm from the previous section for parallelisation. We have chosen OpenCL as the interface for GPU programming as it is an open standard supported by both major vendors of consumer graphics hardware. To process the data using OpenCL (the same is valid also for other APIs, e.g. CUDA) a user has to divide the whole processing task in fragments (groups). Threads for one fragment share some device memory and can be synchronised by a special barrier call. Each thread inside a group receives its coordinates which can be used to localise it inside the group. To provide correct results race conditions should be avoided, i.e., no two threads can write to the same location or read from a location which can be written to by other thread. The common approach to achieve this is to divide the data in blocks and for each block to bijectively assign threads to voxels.

Because of the possible race conditions we cannot use the approach based on labelling all voxels of the steepest descent path of the original algorithm. We proposed two different solutions each modifying the first pass of the algorithm. In both cases one buffer stores the input (unprocessed) slice data and one stores processed data as in-situ processing is not possible.

4.1 A brute-force algorithm

In this modification of the original multi-pass algorithm we let each thread to follow the path of the steepest descent until either a la-
belled voxel is found or the path leads to the next slice. The difference to the original resides in that the found label or pointer is not written to each voxel of the path (the possible source of race conditions), but only the starting voxel is updated. In fact, the downhill path is not stored at all.

This approach performs a lot of redundant operations because their result can’t be shared among the threads without synchronisation. On the other side, all computations are done inside the kernel which is run on the GPU without the need to return to CPU code until the whole slice is processed.

4.2 A dynamic programming approach

In the second alternative we use methods of dynamic programming where the computation consists of a number of iterations. The first one finds fragments of downhill paths of length one and in the subsequent iterations we spread the already gathered information about the downhill path until the final result – complete downhill path – is obtained.

The process is identical to that of [Körbes et al. 2009] with one exception – because we don’t have the whole volume present in the memory we consider downhill path complete also when it points to a next slice. Algorithm advances as follows. A thread examines its voxel and if a local minimum or pointer to a next slice is found it immediately returns. If the voxel is unprocessed (this is possible only in the first iteration) a pointer in the direction of the downhill path to the neighbouring voxel is found and stored. The last case is that a pointer to the current slice is found. In this case the voxel where the pointer points is examined and its value is used for the current voxel. Execution of the kernel is repeated until all voxels fulfil Condition 1. Illustration of the process is depicted in Figure 5.

Each iteration of the kernel prolongs path segments (represented by intermediate pointers – pointers to the same slice) visited by threads by factor of 2. After the first execution the voxels with the intermediate pointers point to the location on the downhill path which is in distance of 1. After the second execution of the kernel the intermediate pointers point to the location in distance of 2 (two path segments of length 1 were composed). Prolongation of the downhill path follows in the similar manner in each subsequent iteration.

To check whether all threads have finished processing we execute a special kernel after a certain number of iterations. In this kernel all unfinished voxels atomically increment a counter. If this counter is non-zero upon the kernel execution we repeat the batch of path prolongation kernel again. We do this until the test kernel returns with zero.

It is apparent that this modification does not perform redundant computations and is therefore more efficient than the brute-force approach. On the other side the whole slice has to be processed before the next iteration of the kernel can be launched, which means the necessity to return to the CPU code. In the brute-force approach the number of iterations and terminal condition is managed from inside the kernel. Execution of the second kernel for detection of unfinished voxels might be time-consuming if there are many voxels which have not been yet fully processed and if the atomic increment operation is a slow one (this is expected) in the used OpenCL driver implementation.

4.2 A dynamic programming approach

In the second alternative we use methods of dynamic programming where the computation consists of a number of iterations. The first one finds fragments of downhill paths of length one and in the subsequent iterations we spread the already gathered information about the downhill path until the final result – complete downhill path – is obtained.

The process is identical to that of [Körbes et al. 2009] with one exception – because we don’t have the whole volume present in the memory we consider downhill path complete also when it points to a next slice. Algorithm advances as follows. A thread examines its voxel and if a local minimum or pointer to a next slice is found it immediately returns. If the voxel is unprocessed (this is possible only in the first iteration) a pointer in the direction of the downhill path to the neighbouring voxel is found and stored. The last case is that a pointer to the current slice is found. In this case the voxel where the pointer points is examined and its value is used for the current voxel. Execution of the kernel is repeated until all voxels fulfil Condition 1. Illustration of the process is depicted in Figure 5.

Each iteration of the kernel prolongs path segments (represented by intermediate pointers – pointers to the same slice) visited by threads by factor of 2. After the first execution the voxels with the intermediate pointers point to the location on the downhill path which is in distance of 1. After the second execution of the kernel the intermediate pointers point to the location in distance of 2 (two path segments of length 1 were composed). Prolongation of the downhill path follows in the similar manner in each subsequent iteration.

To check whether all threads have finished processing we execute a special kernel after a certain number of iterations. In this kernel all unfinished voxels atomically increment a counter. If this counter is non-zero upon the kernel execution we repeat the batch of path prolongation kernel again. We do this until the test kernel returns with zero.

It is apparent that this modification does not perform redundant computations and is therefore more efficient than the brute-force approach. On the other side the whole slice has to be processed before the next iteration of the kernel can be launched, which means the necessity to return to the CPU code. In the brute-force approach the number of iterations and terminal condition is managed from inside the kernel. Execution of the second kernel for detection of unfinished voxels might be time-consuming if there are many voxels which have not been yet fully processed and if the atomic increment operation is a slow one (this is expected) in the used OpenCL driver implementation.

Figure 5: Three iterations of the dynamic algorithm. Lj represent a local minimum label. Numbers next to the arrow represent delta in x and y coordinate where the pointer points.

5 Results

All measurements were performed on an Intel i7 870 CPU with 4GiB of memory and nVidia 460 GTX graphics card with 2GiB of memory. The measured times exclude the time necessary for gradient computation, local minima localisation and labelling. The tests were performed using the Visible Human datasets – head part of the female colour dataset. The volume was cropped to 1400x900x855 (~3.0GiB). For processing in the watershed tool, gradient image in float precision (~4.0GiB) and minima labels in 4-byte integer precision (~4.0GiB) were used. Table 5 shows computational times of presented watershed algorithms on the mentioned dataset.

The used OpenCL interface allows execution of the application on both CPU and GPU. We therefore executed the same program also on CPU processor which provided four cores (eight with multi-threading). This is similar to utilisation of multiple threads by a traditional C++ program.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Execution time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>824</td>
</tr>
<tr>
<td>CPU multi-pass</td>
<td>736</td>
</tr>
<tr>
<td>OpenCL CPU brute-force</td>
<td>276</td>
</tr>
<tr>
<td>OpenCL CPU dynamic</td>
<td>261</td>
</tr>
<tr>
<td>OpenCL GPU brute-force</td>
<td>244</td>
</tr>
<tr>
<td>OpenCL GPU dynamic</td>
<td>244</td>
</tr>
</tbody>
</table>

Table 1: Computation time of various variants of the watershed algorithm. From top to bottom: the original CPU algorithm, multi-pass CPU algorithm and two presented parallel algorithms (OpenCL) executed on CPU and GPU.

We also successfully processed the whole colour female dataset of dimensions 1650x900x5186. Again, input to the algorithm were gradient and minima label volumes of size 28.7GiB each. Result can be seen on the title page.

As it was already mentioned in the previous section checking for unfinished voxels in the dynamic programming version of the algorithm might be a slow operation and is therefore advantageous to perform it only after a certain number of path prolonging kernel iterations.
In each iteration the found downhill path segments are prolonged by factor of two. This means that n iterations are able to follow paths of length up to $2^{n-1}$. By measuring the maximal number of the required iterations for the Visible Human datasets we found out that 7 iterations were sufficient to follow all paths to the terminal voxel. We tested the CT head data set of visible male and the colour data set of the head of the visible female as well. Initial blurring of the data with $\sigma$ values in range $[0.25, 5.0]$ had no effect on the maximum number of the required iterations. This is caused by the logarithmic dependency of the required iterations from length of the downhill path segment. We therefore perform 7 iterations of the prolonging kernel followed by the test kernel for unfinished voxels. The batch of kernel executions is repeated until all voxels pass the execution of the test kernel.

6 Conclusion

In this paper we presented a new algorithm for watershed transform supporting data larger than available memory. One of our previous algorithms designed for CPU was modified for GPU architecture using the OpenCL API. Two variants of the algorithm were designed and implemented. Tests showed that there was no difference in running times indicating that storage of intermediate results in multi-pass algorithms is the major part of the running time and can be shortened by using the RAID technology.

We successfully tested the presented GPU algorithms on the Visible Human project dataset which was larger than the available memory. With the algorithms it was possible to reduce computational time of the watershed transform in respect to the original CPU version approximately 3 times.

7 Acknowledgement

This work was supported by the Slovak Research and Development Agency under the contract No. APVV-20-056105 and Science Grant Agency (VEGA) under contract No. 1/0631/11.

References


